Applications of Divide and Conquer in the Buddy System for Efficient Memory Allocations in Operating Systems

Muhammad Rasheed Qais Tandjung - 13522158
Department of Informatics
School of Electrical Engineering and Informatics
Bandung Institute of Technology, Jalan Ganesha 10, Bandung
13522158@std.stei.itb.ac.id

***Abstract*— The divide and conquer algorithm is a powerful paradigm in computer science, extensively utilized for solving complex problems by breaking them into simpler subproblems. This paper explores the application of the divide and conquer strategy within the context of the Buddy System for memory allocations in operating systems. The Buddy System, known for its efficient memory management capabilities, leverages the divide and conquer approach to handle memory allocation and deallocation by recursively splitting and merging memory blocks. Through theoretical analysis and empirical evaluation, we demonstrate the effectiveness of this combined approach in improving memory allocation speed. The results indicate that applying divide and conquer techniques in memory allocations significantly enhances overall system performance, making it a robust solution for modern operating system memory management challenges.**

Keywords—Buddy System, Divide and Conquer, Efficiency Optimization, Memory Allocation, Operating Systems

I. Introduction

Memory, sometimes referred to as primary storage or random-access memory (RAM), is among one of the most important components of a computer and its operating system. The memory in a computer is the main location for temporarily storing data and instructions during execution of user and system processes. In the realm of computer science and operating systems, efficient memory management is a pivotal factor that significantly influences system performance and resource utilization.

To address these challenges, the divide and conquer strategy emerges as a powerful paradigm. Divide and conquer is a well-established algorithmic technique that breaks a problem into smaller, more manageable subproblems, solves each subproblem recursively, and then combines the solutions to solve the original problem.

The Buddy System operates on the principle of dividing memory into partitions to accommodate memory requests. It divides the memory into power-of-two sized blocks and maintains a list of free blocks of each size. When a memory request is made, the system searches for the smallest available block that can satisfy the request. If a block is larger than required, it is split into two "buddy" blocks, which can be recombined when freed, thereby reducing fragmentation.

The primary motivation behind this research is to explore how the divide and conquer strategy can be applied to memory allocation to overcome its inherent limitations and improve its performance. By decomposing complex memory allocation problems into simpler subproblems, the divide and conquer approach can offer more flexible and efficient memory management solutions.

This research employs a comprehensive approach that combines theoretical analysis, algorithm design, and empirical evaluation. The study begins with an in-depth examination of the existing memory allocation methods, identifying key areas where the divide and conquer strategy can be integrated, mainly using the proposed Buddy System allocation method.

II. Theoretical Framework

A. *Divide and Conquer*

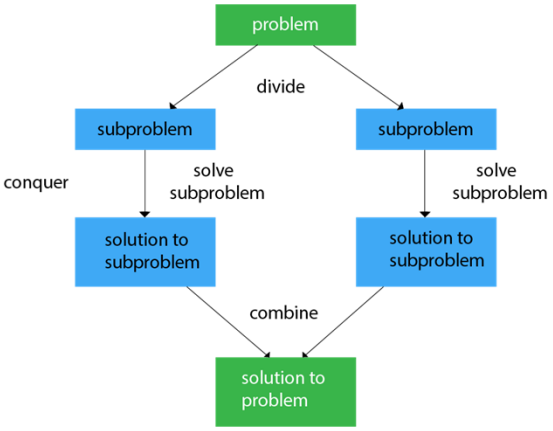

Fig. 1. Illustration of the divide and conquer algorithm.
Source: [2]

The divide and conquer algorithm is a fundamental strategy in computer science and algorithm design, used to solve complex problems by breaking them down into simpler subproblems. The core idea involves three main steps: divide, conquer, and combine.

In the divide step, the original problem is divided into smaller, more manageable subproblems. This division continues recursively until the subproblems reach a base case, which is typically simple enough to be solved directly. The conquer step involves solving these smaller subproblems, often through further recursive application of the divide and conquer strategy.

Once the subproblems are solved, the combine step merges the solutions of the subproblems to form the solution to the original problem. This merging process is crucial as it synthesizes the results of the subproblems into a cohesive solution. The efficiency of the combine step can significantly affect the overall performance of the algorithm.

Divide and conquer algorithms are known for their efficiency, particularly with problems that exhibit recursive substructure and overlapping subproblems. They often lead to algorithms with logarithmic or linearithmic time complexities, making them suitable for large datasets and complex computations. This approach also lends itself well to parallel computing, as the independent subproblems can be solved concurrently, further enhancing performance.

B. CPU Execution and the Memory Hierarchy

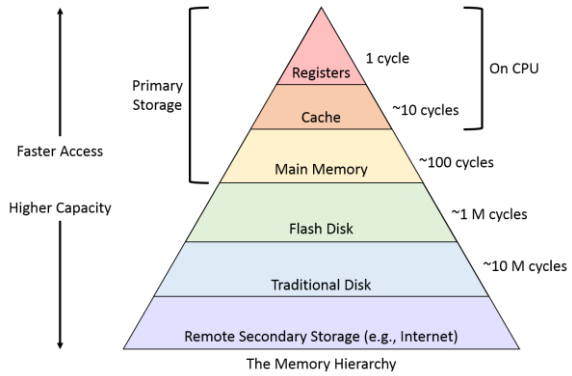

Fig. 2. Diagram of the memory hierarchy.
Source: [3]

In a computer, it is known of three, among many, of main components for computer storage, typically ordered in a pyramid structure known as a memory hierarchy. This structure goes from the very top of the pyramid, which holds the fastest as well as the most expensive types of storage, resulting in small rooms for data, all the way to the bottom which holds the slowest but most spacious types of storage.

The CPU executes millions of instructions per second, and each of these instructions need to be stored in computer storage, else the computer will not know what instruction to execute. Because of the frequency of this action, the speed of which the CPU finishes each execution matters, and a very slight delay in instruction execution time can result in huge performance drops.

The register is the fastest type of computer storage, since it is the one located the closest to the CPU. It is also the smallest type of computer storage, typically only having room for a few tens of kilobytes of data. This poses as a huge problem, since the modern computer typically needs to execute billions to trillions of executions in a short time interval, the volume of which obviously a few kilobytes of storage could not handle.

The external disks, then, poses as a possible solution to this storage problem. An average storage disk is typically able to store up to hundreds of gigabytes, or even terabytes of data. It is, however, a component external to the main computer system, and therefore each access to a single unit of data will require I/O transfer, typically taking enormously longer times to finish relative to a single register access.

Thus the correct solution to this problem involves an intermediary storage device in the middle of registers and external drives in the memory hierarchy. This is where main memory comes in. The main memory is a storage structure that the CPU can access almost as fast as a register, but yet is able to store gigabytes of data, making it the perfect component for storing instructions and data of currently running tasks.

C. Main Memory

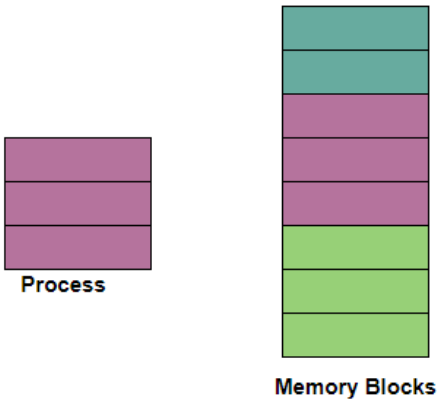

Fig. 3. An illustration of computer memory.
Source: [4]

Computer memory, to which the programmers are concerned, is essentially just a large continuous array of bytes. In this sense, memory holds a huge resemblance to external storage drives (especially SSDs, sometimes also called non-volatile memory nodding towards their similarity). This makes sense in a functional aspect, since main memory is essentialy just a storage device with fast access, typically used for currrent execution instead of long-term data archives.

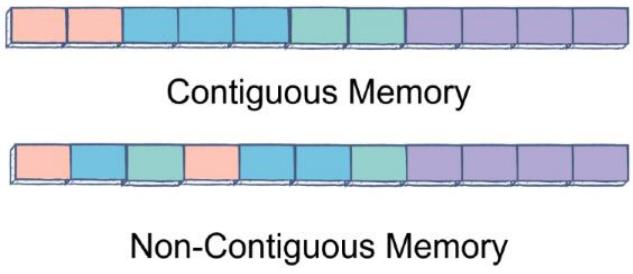

Fig. 4. Contiguous and non-contiguous memory allocation.
Source: [5]

Similar to how files are stored in external storage drives, how the instruction and data of processes are stored in memory can

be in one of two ways, contiguously allocated and non-contiguously allocated, as illustrated in Fig. 3. Both methods have their own advantages, with non-contiguous memory allocation typically being used in most modern computers. This paper will mainly focus on analyzing the efficiency of contiguous memory allocation methods.

One of the most commonly used methods for contiguous memory allocation is the first-fit method, where the memory array is scanned sequentially until a hole large enough to fit a certain process size is found. As expected, this algorithm would have linear time complexity, since at worst-case, the entire memory array would be checked.

D. Buddy System

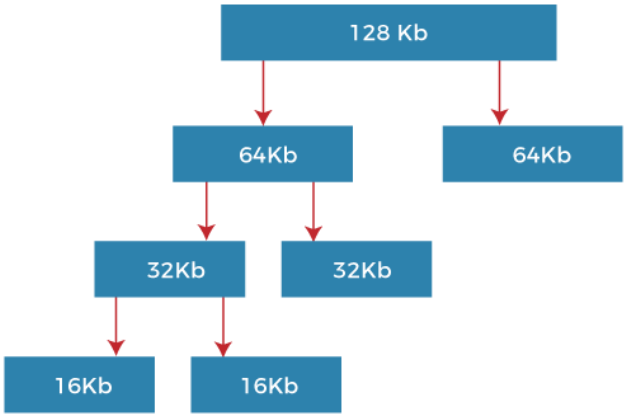

Fig. 5. Illustration of the Buddy System memory allocation method.
Source: [6]

A possible alternative to the first-fit method is the Buddy System, which works on blocks of memory of sizes powers-of-two. The Buddy System starts with one large contiguous block of memory, and for every memory allocation request it gets, it will recursively split the block in half, until the smallest block that is bigger than the requested amount is found.

Different from the sequential search approach used by the first-fit method, the Buddy System uses a divide-and-conquer aproach, where the problem (large block of memory) is continuously split into smaller subproblems (smaller blocks).

III. Methodology

A. Overview of Simulation

With the theoretical frameworks of the divide and conquer algorithm as well as memory allocation and the Buddy System established, this paper will try to verify the efficiencies of divide and conquer in memory allocations using a simulation of memory and processes in Java using threads.

The implemented program will simulate both the standard method of contiguous memory allocation, as well as the Buddy System method using divide and conquer. The reason for implementing both algorithms is for the ability to compare the efficiency of both methods.

The simulation will first start with a pool of processes queueing to be given memory space, each having its own size and runtime duration. The main program places these processes in a queue, and in each iteration one process will be taken from the queue and placed into a slot in memory that is able to fit the size of the process. If there are no slots available that is large enough, the process will be returned to the queue, and the program will try inserting the next process in the queue into memory. This process will keep repeating until the given process queue is empty.

B. The Process Class

Since this simulation will be performing memory allocations on processes, there of course needs to be an implementation of a certain process component. The process, implemented as a class, will be the functional unit of the simulation.

```java
public class Process implements Runnable {
    private final int pid;
    private final int size;
    private final float time;
    private final MemoryInterface memory;
    private final Random random;

    public Process(int pid, int size, float time, MemoryInterface memory) {
        this.pid = pid;
        this.size = size;
        this.time = time;
        this.memory = memory;
        this.random = new Random();
    }
```

Fig. 6. Implementation of the process class.
Source: Personal documentation

Fig. 4 shows the basic structure of the process class used for memory allocations. The attributes of the process class, ones of which are important, are as follows:

- PID: The process ID, functions as metadata of the process, simply for identification purposes. The PID of a process given by the program will be based on the order of the process in the given memory pool.
- Size: The size of the process, in bytes. This size attribute will determine the hole that is allocated to the process in memory. If the hole is too small, it cannot fit the process, and thus the program will check the next hole available, until it finds a hole that fits the process. If the memory currently does not have a hole that is big enough, the process will be inserted back to the end of the process queue.
- Time: The time required for the process to finish its task. This time attribute will determine how long the process lives in memory. The main effect of this attribute is that the longer the duration of the task, the longer the process will consume the allocated memory space, and the longer other processes would have to wait to be able to use the allocated space.
- Memory: This is not an actual attribute of the process itself, but simply a pointer to the memory variable. This is such that the process class is able to call the methods of the memory, specifically for allocation and deallocation.

An instantation of a process can be created using the process constructor, also given in Fig. 4. When a process is created, its PID, size, and runtime duration will be instantly set, and cannot be changed. The numbers for these attributes will be stored in a separate .txt file, which contains all the processes that will be used in a certain experiment run.

```
@Override
public void run() {
    try {
        int runtime = (int) time;
        Thread.sleep(runtime);
    } catch (InterruptedException e) {
        e.printStackTrace();
    } finally {
        memory.deallocate(this);
    }
}
```

Fig. 7. Code for simulating a running process.
Source: Personal documentation

When the program finds a fitting slot in memory and places the process in that memory slot, it will simulate a running process, which in this simulation is simply just creating a new thread that sleeps for the given runtime duration of the process.

The allocation, which is searching for the fitting slot in memory for the process, will be done in the main program and thus is not of concern of the process class. Memory deallocation, though, is of concern, because when the memory is deallocated is based on when the process will finish its task. That event happens after the process thread has slept for the given runtime duration, thus in the run code block for the process, it will call the deallocate method on the memory after it has finished running.

C. *The Memory Interface*

```
public interface MemoryInterface {
    public int allocate(Process process);
    public void deallocate(Process process);
    public void print();
}
```

Fig. 8. Memory interface for both memory allocation implementations.
Source: Personal documentation

Because of the existence of two implementations of memory allocation methods in this simulation, there will be a need to create classes for both methods. But they both should be pretty similar to each other, only differing in its allocation and deallocation algorithms, and thus the main program and processes would need a common interface that can access both classes, else there will be code redundancy introduced.

Thus, Fig. 6. shows the memory interface that will be implemented by both allocation methods. The interface is relatively simple and only includes two main methods, an allocate method and a deallocate method, since these two procedures are mainly where the allocation algorithms differ.

D. *Standard Memory Allocation Method*

```
public class Memory implements MemoryInterface {
    // Buffer for array of contiguous memory
    private final char[] memory;

    // Map of pid to memory location
    private final Map<Integer, Integer> allocated;

    // Constants for memory state
    private static char EMPTY = '.';
    private static char ALLOCATED = 'X';

    // Constructor
    public Memory(int size) {
        memory = new char[size];
        Arrays.fill(memory, EMPTY);

        allocated = new HashMap<>();
    }
```

Fig. 9. Implementation of the memory class.
Source: Personal documentation

After an implementation of a process class, there of course needs to be implementation of a memory class. This class not only holds the contiguous array of memory in the simulation, but also the methods and algorithms for allocation and deallocation of the memory, as given in the memory interface.

The memory class given in Fig.7 will be one of two classes of memory, with this one implementing the standard first-fit algorithm for memory allocation. With this class of memory, processes will be allocated to the first hole that can fit the size of that process.

The contiguous bytes of memory will be stored in the memory attribute of the class, which is implemented as an array of char. Each index of this array can be one of two values, empty or allocated, with empty slots being marked as ‘.’, and allocated slots being marked as ‘X’. Slots of the array that is being used by a process will be marked as allocated, and slots not being used will be marked as empty.

There is also an allocated attribute, implemented as a map, to keep track of how many processes is allocating memory. The key to this map is the PID of a certain process, and the value is the starting address in memory of that process. Thus it can easily be located in what address a certain process resides. This allocated map will be mainly used for deallocation. When a process is finished running and deallocates memory, its PID is used as a key to the map, which will return the starting address of the allocation, and along with the size of the process, the amount of slots to deallocate from memory can be known.

The constructor for the class has as a parameter a size value, which will determine the size of the array of memory in bytes. Once the array has been allocated, all of its slots will be filled with the empty slot value.

```
// Allocate memory for a process
public synchronized int allocate(Process process) {
    int size = process.getSize();
    int start = -1;
    for (int i = 0; i < memory.length; i++) {
        if (memory[i] == EMPTY) {
            int j = i;
            for (; j < memory.length && memory[j] == EMPTY && j - i < size; j++);
            if (j - i == size) {
                start = i;
                break;
            }
        }
    }
```

(a)

```
if (start != -1) {
    for (int i = start; i < start + size; i++) {
        memory[i] = ALLOCATED;
    }
    allocated.put(process.getPid(), start);
}

return start;
```

(b)

Fig. 10. Implementation of the memory allocation algorithm, (a) Finding the appropriate slot, (b) Allocating the slot if found
Source: Personal documentation

Fig. 8 shows the implementation for allocating memory to a process. As is the first-fit algorithm, it will start from the very start of the array, iterating up until it finds a slot that is large enough to store the given process. When the large enough hole is found, it will set all the slots in the range of the size of the process to be allocated, starting from the hole's starting point. The PID and start address pair will also be stored in the allocated map.

```
// Deallocate memory for a process
public synchronized void deallocate(Process process) {
    int start = allocated.get(process.getPid());
    for (int i = start; i < start + process.getSize(); i++) {
        memory[i] = EMPTY;
    }
    allocated.remove(process.getPid());
}
```

Fig. 11. Implementation of memory deallocation

The procedure of memory deallocation of a process is fairly simple. Because of the allocated map already defined previously, deallocating a given process is simply just retrieving its starting address in memory, and iterating as much as the process size, turning all slots from allocated to empty. The allocated process PID will also be removed from the allocated map, hence freeing the slot in memory for other processes to use.

E. Buddy System Memory Allocation Method

```
public class BuddyBlock {
    // Start address of block
    private final int start;

    // Size of block
    private final int size;

    // Is block allocated
    private boolean isAllocated;

    // Child, parent, and buddy blocks
    private BuddyBlock left;
    private BuddyBlock right;
    private final BuddyBlock parent;
    private BuddyBlock buddy;
```

Fig. 12. Implementation of a block in a Buddy System.
Source: Personal documentation

Since the Buddy System is a more complicated memory allocation algorithm, working in blocks of powers of two, implementation of it will be more complicated than the standard algorithm's implementation as well. To ease the process of managing the allocation and deallocations of blocks, a BuddyBlock class would need to be implemented first.

A buddy block class simply represents a block of memory, having a size of a power of two. Defining a class for a block of memory would obviously need to store its starting address and its block size in its attributes, as given in Fig.10. An isAllocated attribute is also defined, to differ between blocks that are being used by a process, and blocks that are not being used.

The main feature of the Buddy System is that memory consists of blocks that can be split into smaller blocks, and smaller blocks that can be merged back into larger blocks. A splitting of a block can be thought of as a block being split into two childrens, one being the left half of the block, and one being the right half of the block. Hence, the buddy block class can be thought of as a binary tree node that can have a left child and a right child.

Thus a buddy block will store its left and right childs in the left and right attributes, respectively, as shown in Fig.10. These children of a certain block will in turn store a reference to its prent as one of its attributes, which will help during the merging process. When two children of a block merges back to its parent, both child blocks need to be empty and not allocated, hence the buddy block class will also keep a reference to its buddy block in its attributes, also for merging purposes.

```java
public class BuddyMemory implements MemoryInterface {
    // Buffer for array of contiguous memory
    private final char[] memory;

    // Root block of memory
    private final BuddyBlock root;

    // Map of pid to memory block
    private final Map<Integer, BuddyBlock> allocated;

    // Constants for memory state
    private static char EMPTY = '.';
    private static char ALLOCATED = 'X';
```

Fig. 13. Implementation of the Buddy memory class.
Source: Personal documentation

The implementation for the Buddy memory class, in structure, is not too far different from the standard memory class. An array of char is still used to represent contiguous bytes of memory, with each slot being either empty or allocated, an a map of allocated processes is also stored. The difference here is that the allocated map stores a Buddy block as its value instead of a starting address, for a certain PID. The class also stores the root block of the memory system, which can be though of as the root node in the binary tree that is the memory blocks.

```java
// Find a free block for a process
private BuddyBlock allocateFindBlock(BuddyBlock block, Process process) {
    // Return null if block is allocated
    if (block.isAllocated()) return null;

    // If block is smallest possible block
    int childSize = block.getSize() / 2;

    if (childSize < process.getSize()) {
        if (block.isLeaf()) return block;
        else return null;
    } else {
        // If block is a leaf, split block
        if (block.isLeaf()) block.split();

        // Recursively find through left and right children
        BuddyBlock left = allocateFindBlock(block.getLeft(), process);
        if (left != null) return left;
        return allocateFindBlock(block.getRight(), process);
    }
}
```

Fig. 14. Implementation of finding a free block in the memory tree.
Source: Personal documentation

The biggest difference in this new Buddy memory class is, of course, the allocation and deallocation algorithms. Though with allocation, the big difference mostly comes from how to find the free block for the process.

Because memory in the Buddy system is treated as a binary tree, searching for a free block transforms from a sequential search problem to a tree traversal problem, which is now solved recursively as shown in Fig. 12. We first determine whether a given block fits to be used or not, and if it is still big enough, we split the block if it's a leaf node, and if it's not a leaf node, we recursively search through its left and right children.

```java
public void deallocateMerge(BuddyBlock block) {
    // Return if block is allocated
    if (block.isAllocated()) return;

    // Return if parent is null (block is root)
    BuddyBlock parent = block.getParent();
    if (parent == null) return;

    // Return if buddy is allocated or not leaf
    if (block.getBuddy().isAllocated() || !block.getBuddy().isLeaf()) return;

    // Merge block with buddy
    parent.merge();
    deallocateMerge(parent);
}
```

Fig. 15. Implementation of merging two blocks.
Source: Personal documentation

Deallocation remains fairly similar to the standard algorithm, with the distinction that if two buddy blocks are completely deallocated, it should be merged back into one block such that larger processes can use those blocks again. The merging of blocks here, just like block searching and splitting, is done recursively to a block's parent until no more merging can be done.

IV. Results and Analysis

With the data structures and algorithms of the implementation already being established, this paper will now begin to run the simulation program to experiment on the efficiencies of both memory allocation methods. We begin by demonstrating how the simulation of both algorithms work to handle a memory of queueing processes.

A. *Running the Simulation*

```
Memory layout: ................................................................
................................................................
................................................................
................................................................
................................................................
................................................................
................................................................
................................................................
................................................................
................................................................
....
Free memory: 1024
```

Fig. 16. An empty memory with no allocated processes.
Source: Personal documentation

Fig. 14 shows a newly created memory with no allocated processes. This memory is at its maximum capacity and can hold 1024 bytes of process data.

```
Process 0 (size 30) is running for 7475 ms
Memory layout: XXXXXXXXXXXXXXXXXXXXXXXXXXXXXX................................
................................................................
................................................................
................................................................
................................................................
................................................................
................................................................
................................................................
................................................................
................................................................
....
Free memory: 994
```

Fig. 17. The first process is allocated to the start of the memory array.
Source: Personal documentation

When the first process in the queue requests for memory, it is allocated to the start of the memory array, since currently the

entire memory is one large hole, and the process is allocated to the start of that hole. The allocation of that process is symboled by the yellow X's in the memory array.

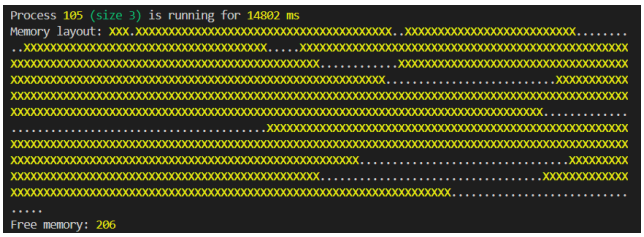

Fig. 18. The state of the memory array after a few seconds
Source: Personal documentation

Processes from the queue will continue to be allocated to memory, after which the process will execute for a certain amount of duration, and then finish and deallocate its memory for other processses to use.

After a large amount of allocating and deallocating, many processes will be scattered around the memory array with lots of holes in-between two processes, indicating a previous process that has finished executing and deallocated its memory. The illustration of such a state is given in Fig.16.

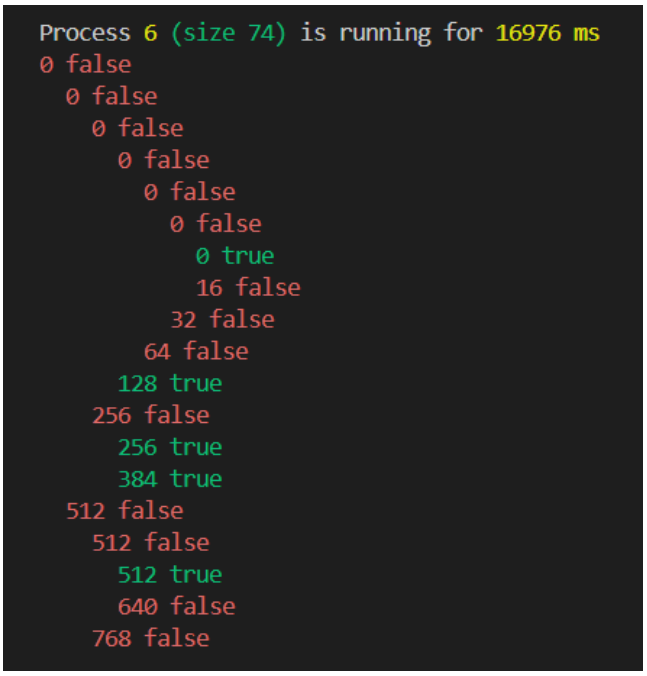

Fig. 19. Allocations of blocks in the block tree of the Buddy allocation system.
Source: Personal documentation

The behavior of the Buddy memory allocation system when given a pool of queueing processes is more or less the same, with the key difference being that processes are only allocated starting at addresses of powers-of-two. Fig. 17 showcases how the block tree is generated given requests for processes, with green nodes indicating that they are allocated to processes, and red nodes indicating that they are free. Notice how green nodes are always at the leaf nodes, since it is impossible for a block to be allocated and yet have children blocks at the same time.

B. Testing the Efficiencies of Both Algorithms

We are now ready to test the efficiencies of allocation for both algorithms. The mechanisms of the test are as follows:

- Given a pool of 500 processes, the program will test both algorithms for allocating memory to all of these processes until the pool is empty.
- The processes will have varying values of size and runtime duration. These values have been generated beforehand and stored in a .txt file, thus testing for both algorithms will use the same pool of processes for a given run.
- For every process, the amount of time taken to allocate memory for it will be taken into account. The average time taken for both algorithms will be compared in the end of the experiment.

Given these conditions, for a set memory buffer size of 8192 bytes, and running the experiment on 10 times each with its own pool of randomly generated processes, the following data for average time allocation is discovered.

TABLE I. AVERAGE ALLOCATION TIME OF BOTH ALGORITHMS

Test No.	Average Allocation Time (ns)		
	Standard	***Buddy***	***Diff.***
1	34.561	17.283	17.278
2	44.512	19.425	25.087
3	32.561	23.415	9.146
4	46.541	24.331	22.210
5	49.813	21.241	28.572
6	37.424	27.382	10.042
7	39.414	29.213	10.201
8	41.398	17.325	24.073
9	44.324	19.419	24.905
10	38.321	22.214	16.107

As shown in Table I, in all experiments it is shown that the Buddy system has faster allocation times than the standard allocation system. The difference between the standard allocation times and the Buddy allocation times has an average of 18.762 ns, implying that based on the test results, the Buddy allocation system will be 18 nanoseconds faster in memory allocation than the standard first-fit memory allocation system.

This makes sense when we consider the types of algorithms that these methods use. The first-fit algorithm can be thought of as a sequential search in an array, typically having O(n) time complexities. The Buddy System, meanwhile, uses a divide and conquer approach, typically having O(log n) time complexities.

It should be noted, though, that the unit of time used in these experiments is the nanosecond (ns), a very small unit of measurement. This experiment is, however, about the time taken for a single memory allocation. In a typical computer system used day-to-day, a computer might need to complete millions of memory allocations per second, and thus the very small time differences will slowly build up into huge performance differences.

CONCLUSION

From the results of the experiment on algorithm efficiency, it has been shown that the Buddy System performs memory allocations more efficiently than the standard memory allocation algorithm. This proves that a different choice of algorithm can bring a huge impact on efficiencies of solutions. Memory allocation using the Buddy System is just one example of the applications of the divide and conquer algorithm to increase efficiency in solving problems. The reasonable takeaway from this study is that a deep understanding of algorithms is an important aspect of computer science and related fields.

ACKNOWLEDGMENT

The author would like to deeply thank Mr. Dr. Ir. Rinaldi Munir, M.T., and Mr. Monterico Adrian, S.T., M.T. as the author's lecturers of the Algorithmic Strategies course, and by extension, the entire Algorithmic Strategies staff, consisting of lecturers and assistants, for giving the author a chance to not only deepen their knowledge of the field, but to conduct this study as well. Lastly, but certainly not least, the author would like to thank their friends and families, for always giving them support and always being present while going through every hardship experienced in the process of conducting their study as well as writing this academic paper.

REFERENCES

[1] Stallings, W. (2013). Operating Systems: Internals and Design Principles, Seventh Edition. Pearson Education Limited.

[2] *Divide and Conquer Introduction*, https://www.javatpoint.com/divide-and-conquer-introduction. Accessed June 12th 2024.

[3] *Memory Hierarchy*, https://www.cs.swarthmore.edu/~kwebb/cs31/f18/memhierarchy/mem_hierarchy.html. Accessed June 12th 2024.

[4] What is Contiguous Memory, https://awaitdeveloper.medium.com/what-is-contiguous-memory-222f58f28079. Accessed June 12th 2024.

[5] Arrays vs. Lists, https://alirezafarokhi.medium.com/array-vs-list-compare-array-and-list-performance-in-c-722316603c8c. Accessed June 12th 2024.

[6] What is Buddy System, https://www.javatpoint.com/what-is-buddy-system/. Accessed June 12th 2024.

STATEMENT

I hereby declare that this paper I have written is my own work, not a translation or adaptation of someone else's paper, and is not plagiarized.

Bandung, 12 Juni 2024

Muhammad Rasheed Qais Tandjung
13522158